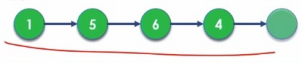
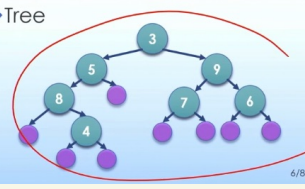
2 Recursive Data Structures

- List



- Tree



6/8/2022

3 Recall: Linked List Data Representation

Assume int elements for now.

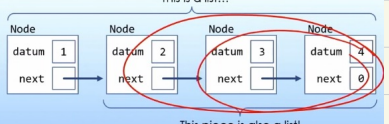
```
struct Node {
  int datum;
  Node *next;
};
```

Used to store an element of the list.

Contains the address of the next node in the list.

- The node structure of a linked list is self-similar!

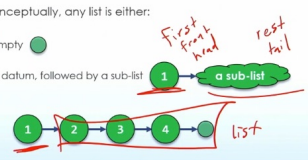
This is a list…



This piece is also a list!

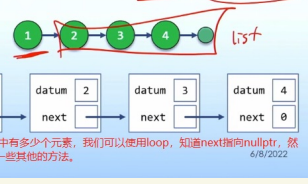
4 Recursively Defining a List

- Conceptually, any list is either:
  1. empty
  2. A datum, followed by a sub-list



First front head

rest tail

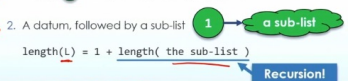


想要知道一个list中有多少个元素，我们可以使用loop，知道next指向nullptr，然而，我们还有一些其他的方法。

6/8/2022

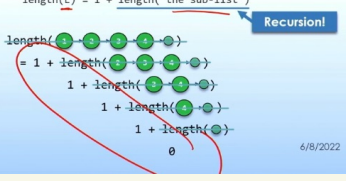
5 Processing a List Recursively

- For example, let's **compute the length of a list L**.
- Consider the two cases:
  1. empty — easy enough: `length(L) = 0`
  2. A datum, followed by a sub-list



`length(L) = 1 + length( the sub-list )`

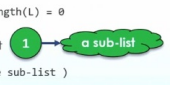
Recursion!



6/8/2022

6 Processing a List Recursively

- For example, let's compute the length of a list L.
- Consider the two cases:
  1. empty — easy enough: `length(L) = 0`
  2. A datum, followed by a sub-list



`length(L) = 1 + length( the sub-list )`

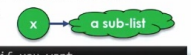
```
int length(Node *node) {
  if (node == nullptr) { // BASE CASE
    return 0;
  }
  else { // RECURSIVE CASE
    return 1 + length(node->next);
  }
}
```

This could also be written as !node.

8 Exercise: max

```
struct Node {
  int datum;
  Node *next;
};
```

- Now, write a function to find the maximum element.
- Hint: You may need to use a different base case.
- Hint: Use the recursive leap of faith on the sub-list.




```
// Use this helper function if you want
int max(int x, int y) {
  if (x > y) { return x; }
  else { return y; }
}
```

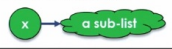
```
// REQUIRES: 'node' must not be null (i.e. the list
//           starting at 'node' may not be empty)
// EFFECTS:  Returns the largest element in the list.
int max(Node *node) {
  // YOUR CODE HERE
}
```

Slides

9 Solution: max

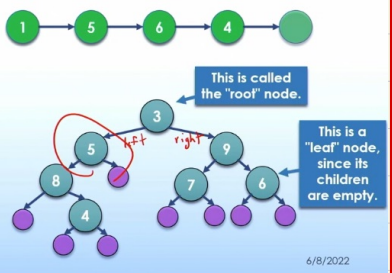
```
struct Node {
  int datum;
  Node *next;
};
```

- Now, write a function to find the maximum element.
- Hint: You may need to use a different base case.
- Hint: Use the recursive leap of faith on the sub-list.




```
// REQUIRES: 'node' must not be null (i.e. the list
//           starting at 'node' may not be empty)
// EFFECTS:  Returns the largest element in the list.
int max(Node *node) {
  // BASE CASE - A single element list
  if (node->next == nullptr) {
    return node->datum;
  }
  else { // RECURSIVE CASE
    return std::max(node->datum, max(node->next));
  }
}
```

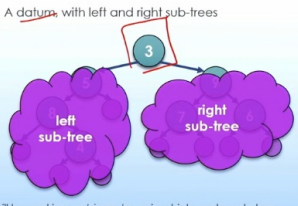
10 Recursive Data Structures



6/8/2022

11 Recursively Defining a Tree

- Conceptually, a tree is either:
  1. empty
  2. A datum with left and right sub-trees

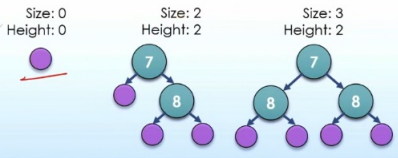


We will be working on binary trees in which each node has two children.

6/8/2022

12 Properties of Trees

- We can measure a tree in two ways:
  - Size: The total number of elements
  - Height[1]: The longest chain of nodes from root to leaf.



1 This is sometimes called the "maximum depth" of the tree.

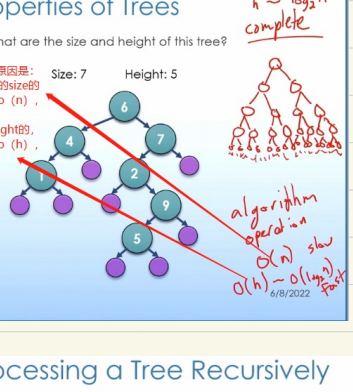
6/8/2022

Slides

13 Properties of Trees

- What are the size and height of this tree?

我们研究size和height的原因是：
1.如果某种算法基于tree的size的那么他的运算时间将会是o（n），较为缓慢。
2.如果某种算法是基于height的，那么他的运算时间将会是o（h），也就是o（log2n）较快。

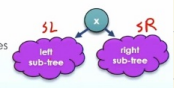


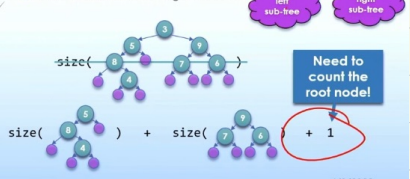
6/8/2022

14 Processing a Tree Recursively

- For example, let's **compute the size of a tree T**:
- Consider the two cases:
  1. empty
  2. A datum, with left and right sub-trees

将左右两边的分支进行抽象，列出表达式。



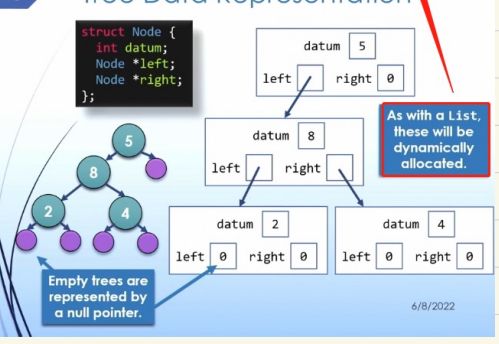


6/8/2022

我们希望每个node的lifetime都是相互独立的，因为我们可以对立面的任意一个node进行删除、更改等操作。

15 Tree Data Representation

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```



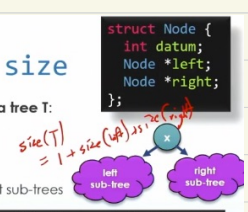
6/8/2022

Slides

16 Example: Tree size

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Let's **compute the size of a tree T**:
- Consider the two cases:
  1. empty
  2. A datum, with left and right sub-trees



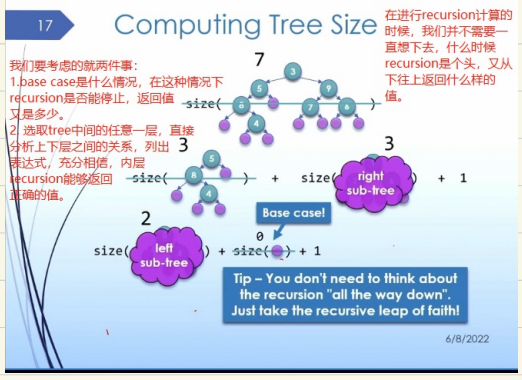

```
// EFFECTS: Returns the size of the tree rooted at 'node'.
int size(Node *node) {
  // BASE CASE - Empty tree has size 0
  if (!node) {
    return 0;
  }

  // RECURSIVE CASE
  return 1 + size(node->left) + size(node->right);
}
```

17 Computing Tree Size

我们要考虑的就两件事：
1.base case是什么情况，在这种情况下recursion是否能停止，返回值又是多少。
2.选取tree中间的任意一层，直接分析上下层之间的关系，列出表达式，充分相信，内层recursion能够返回正确的值。

在进行recursion计算的时候，我们并不需要一直想下去，什么时候recursion是个头，又从下往上返回什么样的值。



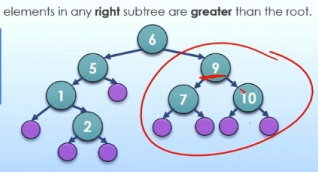
6/8/2022

28 Binary Search Trees (BSTs)

- A tree is a binary search tree if…
  - It is empty

  OR
  - The left and right subtrees are binary search trees.
  - All elements in any **left** subtree are **less** than the root.
  - All elements in any **right** subtree are **greater** than the root.

It is so called because searching for elements can be done efficiently.

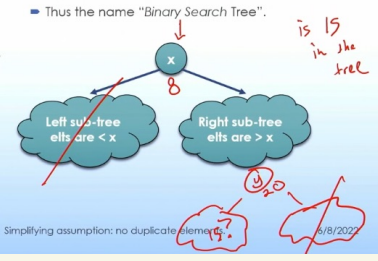


Note: In the slides and on project 5, we make a simplifying assumption that there are no duplicates in our BSTs.

6/8/2022

29 Searching in a Binary Search Tree

- If we're looking for an element in a BST, comparing with the root tells us where to look.
  - Thus the name "*Binary Search* Tree".

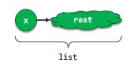


Simplifying assumption: no duplicate elements.

6/8/2022

Exercise 19.1 (bit.ly/3fQE21a)

Determine base cases and recurrence relations for the list functions in the table below. Use terms in the picture at right. Some examples are given to help you get started.



This exercise is challenging! Recursion is a totally different kind of thinking than we use in our normal lives. We promise it gets easier. If you get stuck, check the walkthrough video.

length(list) Finds the # of elements in `list`.	Base Case:	`if empty, length(list) = 0`
	Recurrence Relation:	`length(list) = 1 + length(rest)`
sum(list) Finds the sum of values in `list`.	Base Case:	`If empty, sum(list) = 0;`
	Recurrence Relation:	`sum(list) = list->datum + sum(list->rest)`
last(list) Finds the last element in `list`. REQUIRES: `list` is not empty	Base Case:	`if rest is empty, last(list) = x (Base case is a single element list)`
	Recurrence Relation:	`last(list) = rest(list->rest)`
count(list, n) Counts # of times `n` appears in `list`.	Base Case:	`If empty, count(list,n) = 0;`
	Recurrence Relation: (Hint: Use an if-else here!)	`If (list->datum ==n){ Return 1+count(list->rest,n); } Else{ Return count(list->next,n); }`
max(list) Finds the max value in `list`. REQUIRES: `list` is not empty Hint: use a helper function `max(int,int)`	Base Case:	`If (list->rest == nullptr){ Return list->datum }(only element, must be the max)`
	Recurrence Relation:	`Return max(list->datum, max(list->next));`

Exercise 19.2 (bit.ly/3fQE21a)

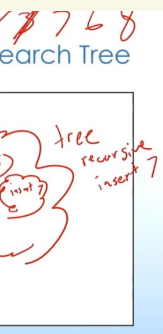
Determine base cases and recurrence relations for the list functions in the table below. Use terms in the picture at right. Some examples are given to help you get started.

This exercise is challenging! Recursion is a totally different kind of thinking than we use in our normal lives. We promise it gets easier. If you get stuck, check the walkthrough video.

size(tree) Finds the # of elements in `tree`.	Base Case:	`if empty, size(tree) = 0`
	Recurrence Relation:	`size(tree) = 1 + size(left) + size(right)`
height(tree) Finds the # of elements in `tree`. Hint: use a helper function `max(int,int)`	Base Case:	`If empty, return 0;`
	Recurrence Relation:	`Return 1 + max(height(tree->left),height(tree->right))`
sum(tree) Finds the sum of values in `tree`.	Base Case:	`If empty, return 0;`
	Recurrence Relation:	`Return tree->datum + sum(tree->left) + sum(tree->right)`
num_leaves(tree) Finds the number of leaf nodes in `tree`.	Base Case: (HINT: there are two base cases here!)	`If (!tree){return 0} If (!tree->left && !tree->right){return 1}`
	Recurrence Relation:	`num_leaves(tree) = num_leaves(left) + num_leaves(right)`
contains(tree, n) Returns true if `tree` contains the value `n`, and false otherwise.	Base Case: (HINT: there are two base cases here!)	`if(!tree){return false} if(tree->datum == n){return true}`
	Recurrence Relation:	`If (contains(tree->left,n)||contains(tree->right, n)) return 1; Else return 0;`

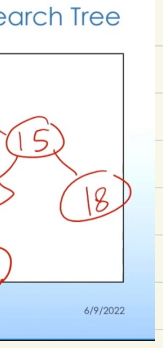
Slides

30 Building a Binary Search Tree



30 Building a Binary Search Tree

需要保证root左边的tree里面的数字都比他小。



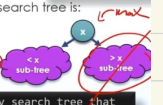
6/9/2022

Slides

31 Example: BST max

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- The maximum element in a binary search tree is:
  1. The datum in the node if the right sub-tree is empty
  2. Otherwise, the maximum element in the right sub-tree




```
// REQUIRES: 'node' must be a binary search tree that
//           is not empty (i.e. 'node' is not null)
// EFFECTS:  Returns the largest element in the tree.
int max(Node *node) {
  // BASE CASE - Right sub-tree is empty
  if (!node->right) {
    return node->datum;
  }
  else { // RECURSIVE CASE - Right sub-tree not empty
    return max(node->right);
  }
}
```

tail recursive

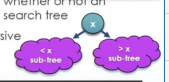
32 Exercise: BST contains

两个base case
1.找到
2.没找到

两个recursion
1.左边
2.右边

- Write a function that determines whether or not an element is contained in a binary search tree
- Your solution should be tail recursive

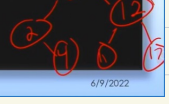
```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```




```
// REQUIRES: 'node' must be a binary search tree
// EFFECTS:  Returns whether or not the tree contains
//           the given item.
bool contains(Node *node, int item) {


  // YOUR CODE HERE

}
```



6/9/2022

34 The BinarySearchTree Interface

BST 1 Map 2 PirateClassifier 3

```
template <typename T>
class BinarySearchTree {
public:
  BinarySearchTree();
  BinarySearchTree(const BinarySearchTree &other);
  BinarySearchTree & operator=(const BinarySearchTree &other);
  ~BinarySearchTree();

  bool empty() const;
  int size() const;
  bool contains(const T &item) const;
  void insert(const T &item);

private:
  struct Node {
    T datum;
    Node *left, *right;
  };

  Node *root;
};
```

remove

6/9/2022

33 Solution: BST contains

- Write a function that determines whether or not an element is contained in a binary search tree
- Your solution should be tail recursive

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

因为是binary search 所以我们左边还是右边是"or"的关系。而下需要同时验证。

```
// REQUIRES: 'node' must be a binary search tree
// EFFECTS:  Returns whether or not the tree contains
//           the given item.
bool contains(Node *node, int item) {
  if (!node) {
    return false;
  } else if (node->datum == item) {
    return true;
  } else if (node->datum > item) {
    return contains(node->left, item);
  } else {
    return contains(node->right, item);
  }
}
```

OR

这个function是tail recursion，因为字符简化虽然有两个contain的recursion，但是我们最终还只会call其中的一个

6/9/2022

35 BinarySearchTree Implementation

- We can implement the functions that operate on Nodes as private, static member functions

```
template <typename T>
class BinarySearchTree {
public:
  bool contains(const T &item) const {
    return contains_impl(root, item);
  }
private:
  Node *root;
  static bool contains_impl(Node *node, const T &item) {
    if (!node) {
      return false;
    } else if (node->datum == item) {
      return true;
    } else if (node->datum > item) {
      return contains_impl(node->left, item);
    } else {
      return contains_impl(node->right, item);
    }
  }
};
```

provided

root inside this

You

和之前所学的static member variable一样，这些static 的private member function只会class中share，而不会在具体的class在local中所创建的instance的object中share。所以这些static function中就没有"this"这也避免了在function 中错误的call root的问题。

26 Types of Recursion

Tree

Time O(h) O(log n)

space O(1)

Time O(n)

- A function is ***linear recursive*** if it makes at most one recursive call each time the function is called.
  - Example: `fact`, `List max`

special case

- A function is ***tail recursive*** if it is linear recursive and all recursive calls are tail calls, so that no work is done after a recursive call.
  - Example: `fact_tail`, `max_tail`

reverse tree, max on BST contains on BST

- A function is ***tree recursive*** if it might make more than one recursive call each time the function is called.
  - Example: `Tree size`, `height`
  - A function doesn't have to operate on trees to be tree recursive! It is tree recursive if the structure of the recursive calls *branches* and thus looks like a tree.

如果是一个tree 并且是tree recursive，那么时间就和tree中的elements的数量成正比

如果是tree并且是linear recursive, 那么就和tree的高度成正比

6/9/2022

35 BinarySearchTree Implementation

BinarySearchTree <int> t; <string> <Cord> <Duck>

- We can implement the functions that operate on Nodes as private, static member functions

然而有些时候，比较大小的operator对于比较的对象会是没有意义的，比如说duck，所以我们就要给这个class新加一个comparator，然后这自己的less（）方程。

```
template <typename T>
class BinarySearchTree {
public:
  bool contains(const T &item) const {
    return contains_impl(root, item);
  }
private:
  Node *root;
  static bool contains_impl(Node *node, const T &item) {
    if (!node) {
      return false;
    } else if (node->datum == item) {
      return true;
    } else if (node->datum > item) {
      return contains_impl(node->left, item);
    } else {
      return contains_impl(node->right, item);
    }
  }
};
```

Comparator

param

less

less(item, node->datum)

static means the function can't access member variables. It shouldn't! The pointer to the rool of the node structure is passed in.

27 Subproblem Graph: Fibonacci

Tree 1 1 2 3 5 8 13 ...

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  return fib(n - 1) + fib(n - 2);
}
```


some probelm is tree recursion problem, but has nothing to do with the tree.
但是在递归过程当中所画出的图形关系就是树状图，这也是tree recursion名字的由来。

Fibonnaci can also be computed iteratively or tail recursively. 6/9/2022

25 Tree height

size loop stack LLRL

- Question: Can the height function be implemented tail recursively? If so, how? If not, why not?
  - Nope! You need to check both sides of the tree, which requires two recursive calls. They can't both be tail calls.

```
// EFFECTS: Returns the height of the tree rooted at 'node'.
int height(Node *node) {
  // BASE CASE - Empty tree has size 0
  if (!node) {
    return 0;
  }

  // RECURSIVE CASE
  return 1 + std::max(height(node->left),
                      height(node->right));
}
```

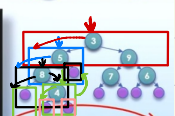
tail recursion == loop

6/9/2022

23 Tree print

```
struct Node {
  int datum;
  Node *left;
  Node *right;
};
```

- Write a function to print the elements of a tree.
- Consider the two cases:
  1. empty
  2. A datum, with left and right sub-trees




```
// EFFECTS: Prints the elements of
//          the tree rooted at
//          'node', with a space
//          after each element.
void print(Node *node) {
  if (node) { // RECURSIVE CASE
    cout << node->datum << " ";
    print(node->left);
    print(node->right);
  }
}
```

pre order

Prints 3 5 8 4 9 7 6

6/9/2022

24 Tree Traversals

3 5 9 12 15 17

- For `print()`, we have a choice of when to process a datum
- A ***preorder traversal*** processes a datum before the recursive calls.

Prints 3 5 8 4 9 7 6

```
if (node) { // RECURSIVE CASE
  cout << node->datum << " ";
  print(node->left);
  print(node->right);
}
```

- An ***inorder traversal*** processes a datum between the calls.

Prints 8 4 5 3 7 9 6

```
if (node) { // RECURSIVE CASE
  print(node->left);
  cout << node->datum << " ";
  print(node->right);
}
```

BST

less

greater

- A ***postorder traversal*** processes a datum after the recursive calls.

Prints 4 8 5 7 6 9 3

```
if (node) { // RECURSIVE CASE
  print(node->left);
  print(node->right);
  cout << node->datum << " ";
}
```

6/9/2022